Software Design of Energy-Aware Peripheral Control for Sustainable Internet-of-Things Devices

Michael Uelschen
University of Applied Sciences
Osnabrueck, Germany
m.uelschen@hs-osnabrueck.de

Marco Schaarschmidt
University of Applied Sciences
Osnabrueck, Germany
m.schaarschmidt@hs-osnabrueck.de

Abstract

The resource-efficient development of technical devices is one of the most important non-functional requirements regarding global warming. This applies especially to the growing field of the (Industrial) Internet of Things. The energy consumption of such systems must be minimized to ensure a long operational lifetime. The realization requires exploiting the possibilities of the complete system (microcontroller and external peripherals) by the software application. However, software engineers are often unaware of the energy-saving properties of the hardware platform. This paper introduces a novel software framework that aims to bridge the gap between the hardware level and the application level. It enables vertical control, i.e., consistent access across multiple software architectural layers. This paper describes the framework in terms of design patterns, shows an implementation based on the C++20 standard, and concludes with an evaluation using a popular hardware platform.

1. Motivation

Over the past few years, the number of applications using Internet of Things (IoT) and Industrial Internet of Things (IIoT) based solutions has been increased. Typical domains for IoT and IIoT are environmental monitoring [1, 2], smart cities (e.g., air and noise monitoring, smart parking) [3], smart factories [4], and smart agriculture (e.g., precision farming, animal monitoring) [5, 6]. Due to this expansion, it is expected that the number of IoT devices will reach 43 billion by the year 2023 [7]. An analysis performed by Cisco [8] stated that IoT devices sending data to IoT edge devices and IoT cloud solutions will account for 50 percent of all networked devices in the same year. The complexity is increased by the heterogeneous nature of IoT due to the variety of processor architectures, sensors, actuators, communication interfaces, operating systems, and driver implementations.

From an energy-efficient system design perspective, IoT devices are the most critical part of a typical IoT architecture. The supply of power is a major challenge, especially for battery-powered devices with an expected operational lifetime between weeks and decades. Additionally, if those devices are placed in harsh environments or are buried underground, maintenance (e.g., recharging) is not possible or results in higher costs. Energy harvesting capabilities can extend the operational lifetime but tend to be an unreliable source of energy. While energy efficiency is well-addressed by designers and researchers at the hardware level [9], it is often neglected by software developers, which are in many cases unaware of how energy-efficient software is specified [10, 11]. While background activities and unnecessary resource usage are among the main reason for energy consumption problems [10], energy consumption needs careful consideration from the software perspective [12] because up to 80 percent of the system's total energy consumption is caused by the interaction between software and hardware [13]. Therefore, software plays an important role when addressing the energy efficiency of an embedded system or IoT device. Optimization techniques on the software level can reduce the total energy consumption, extend the operational lifetime, influence the overall hardware design, and therefore reducing the costs.

Our approach is focused on optimizing the interaction between software and hardware (e.g., the behavior of the microcontroller (MCU) and connected devices) by providing a framework as a link between the software application layer and hardware devices. The rest of this paper is organized as follows: Section 2 presents the contributions of this paper. Related work is presented in Section 3. By the classification of hardware devices in Section 4, we define requirements and properties which are used for

URI: https://hdl.handle.net/10125/80275
978-0-9981331-5-7

energy management. In Section 5, the policy-based device pattern as the first part is present and in Section 6, we introduce the energy-aware control design as the second main part of the framework. The evaluation of our approach is described in Section 7. Section 8 concludes our paper. Our presentation follows UML [14] as a modeling language.

2. Our Contribution

In this paper, we present an innovative and coherent framework for the development of energy-aware software. The framework takes the behavior of the MCU as well as connected devices (e.g., external sensors and actuators) into account. In detail, we present the following novel contributions in this paper:

- We introduce a new approach to model the device driver layer controlling external devices (Section 5). A policy-based access pattern is achieved by inverting the dependencies to the underlying hardware layer. Applying this principle increases the flexibility and portability and minimizes the effort when modifying the hardware platform.
- Based on this device pattern, we derive an energy-aware framework (main contribution) controlling the power modes of the connected devices (off-chip) and the internal hardware units of the MCU (on-chip). Using novel patterns as the PowerLock and the PowerMonitor [15], we can control the power consumption based on the functional needs of the software application (Section 6).
- We evaluated and proved our framework using the C++ programming language. The new C++ standard C++17 and especially the recently released C++20 version allow the definition of requirements of template types, called concepts.

We conclude our contribution by giving promising results of measurement on a popular hardware platform designed for the IoT domain (Section 7). In a typical use case, more than 20% of the energy can be saved by using the outlined energy-aware framework.

3. Related Work

The authors in [16] analyzed the effect of different well-known design patterns (e.g., Template Method, State, and Strategy) described in [17] on energy consumption and implemented alternative nonpattern solutions for comparison. Other approaches analyzed the impact of object-oriented programming features (e.g., polymorphism, inheritance, and operator overloading) and design patterns that use the features on energy efficiency while focusing on server systems [18] and embedded systems [19]. In summary, the related work presented above focuses on aspects of programming languages, e.g., memory management, as well as memory usage and the overall execution time. Changes to the software applications on this level only affect the performance of the MCU. With the emergence of ultra-low-power MCUs in energy-efficient embedded systems that require little processing power, the MCU only plays a minor role while peripheral devices like sensors and communication interfaces constitute a significant part of the overall energy consumption of the system. In [20], a concept for an energy-aware device driver is presented, which can trace and map the current state of the hardware component into a driver representation. While this concept can be used for analysis purposes, it does not provide an access policy to prevent misusage nor actively influence the energy consumption of the hardware component.

Our approach focuses on the optimization of energy efficiency for these peripheral devices by targeting software-hardware interactions at the behavior level of an application, where changes in the control flow can result in a more energy-efficient system. For example, in case peripheral devices like sensors are misconfigured or misused, e.g., being active even if they are not required to, the power consumption of the system is going to be higher than for a variant of this system in which these devices are properly configured and used.

4. Requirements

Due to the variety of microprocessors and the large number of external devices in the market such as sensors or actuators, it is very challenging or even impossible to find or define a common abstraction of functions and properties for these devices. Additionally, there is also no cross-manufacturer strategy like Autosar [21], which is mainly used in the automotive domain. For this reason, we take the opposite and more beneficial approach. We define a set of classes with different requirements and properties according to energy management capabilities. Real hardware devices can be categorized into one of the four following classes:

- Class 0: A device that does not offer any possibilities to regulate the power consumption through the application. The only option is to switch the device on or off externally, for example by a mechanical switch, which must be included in the circuit design.
- Class 1: A device that does not include its own energy management unit, but whose energy consumption can be controlled via the software

application. Such a device thus has at least two different energy states.

- Class 2: A device that implements its own internal logic (realized in hardware) to control energy consumption. Through the application, it is possible to switch between energy states. The functionality of the device and the quality of service (e.g., precision, number of measurements) depends on the selected energy state.
- Class 3: A device in this class supports the requirements of the previous classes 1 and 2. The combination allows sophisticated energy management strategies to be implemented.

Our proposed policy-based driver pattern is agnostic to this classification and can be used independently. The presented energy-aware framework (see Section 6) supports classes 1-3 which can be controlled with respect to their power consumption. In addition, the following constraints must be considered:

- Con-1: The control of power consumption for hardware devices strongly depends on the software application. The overall control flow is defined at a higher level and therefore the knowledge of how and when hardware devices are used. Lower software layers must support this and should not implement an automatism.
- Con-2: The change of energy states and the energy-aware behavior of devices are driven by the system state. Thus, the system context influences the required wake-up and sleep behavior.
- Con-3: Specific devices, such as radio interfaces, must be activated regularly, otherwise the established connection will be reset. This results in additional energy consumption, which must be spent to re-establish the connection.
- Con-4: The framework must take dependencies between different (hardware) devices into account when changing from one energy state to another. For example, a communication bus can only be set to sleep mode when all connected nodes have stopped communicating previously.

5. Policy-based Device Pattern

After an introduction, the following section describes a pattern for the development of device drivers. This strategy-based pattern follows the Dependency Inversion-Principle (DIP) as one of the five SOLID principles in object-oriented design by Robert C. Martin [22]. Although the approach has been known for a long time, to our knowledge, it has not been consistently used in device driver design. Instead, a hierarchical approach with a hardware-oriented abstraction layer is predominant here.

5.1. Introduction

There are some attempts to standardize access to connected devices. One of the widely used programming interfaces for UNIX-like operating systems is POSIX I/O [23], which allows stream- or file-like access to individual devices essentially via five functions (open, close, read, write, ioctl). A standardized interface for the control of the energy consumption is not considered but has to be realized via specific ioctl calls. CMSIS [24], a programming interface driven by ARM for their processor platforms, goes one step further. In addition, there is a function for controlling the energy consumption of the corresponding internal hardware unit (on-chip). Popular community-based initiatives such as Arduino [25] and Mbed [26] offer a hardware abstraction layer (HAL) for the supported platforms, but without a standardized interface for power consumption control. All the aforementioned approaches only consider the MCU and not the system including connected, external devices.

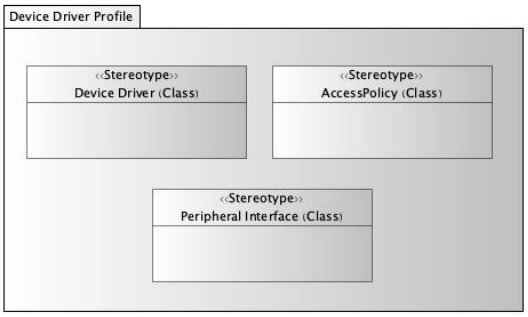

Figure 1. Device driver profile

The structure of the following sections is based on the de facto standard format for a uniform description of design pattern, described in [17].

5.2. Abstract and Problem Description

The access from the application layer to external peripherals is vertical through a stack of different software layers. The inversion of dependencies minimizes the effort for changes to the hardware platform. This design pattern is the basis for controlling power consumption in the following chapter.

To control external devices, e.g., to query sensors, a lower hardware layer must be accessed. Since devices can be physically connected in different ways, an additional abstraction becomes necessary to minimize efforts and costs for later changes of the hardware configuration.

5.3. Pattern Structure

In contrast to a hierarchical approach of device drivers, we follow the DIP. We introduce an additional abstraction to describe the required interface accessing the external peripheral device. By following this approach, we can address, among others, the following use cases:

- The peripheral device may support different hardware interfaces. For example, the Bosch BME280 sensor [27] can physically be connected as an I2C device or using a SPI connection. Although the physical wiring and communication protocols are different, the actual device driver functionality should be independent of them.
- The effort to modify any application logic is minimized or non-existent when the hardware configuration is changed. For example, a LED is directly connected via a GPIO pin. In another situation, the LED is connected by a bus system, e.g., I2C. From the application point of view, these details should be hidden.

The DIP can be realized by the policy-based design method which usually is associated with the C++ programming language [28]. We bring more details on the implementation aspects in the subsequent sections.

Figure 1 shows the basic abstractions as a UML profile diagram using the stereotype notation [14]. The Device Driver class describes an entity representing the functional interface of a peripheral device, which is usually off-chip. The class uses a policy to access such devices. The Access Policy defines the methods to access the hardware required by a Device Driver. The Peripheral Interface class hides the details of the on-chip hardware which is usually memory-mapped.

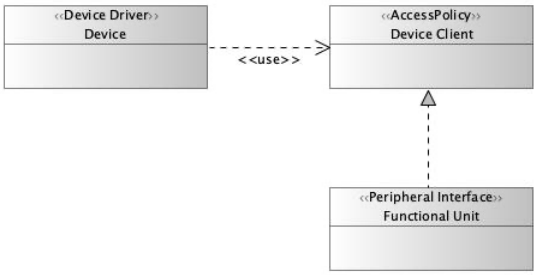

Figure 2. Policy-based device pattern

Figure 2 shows the interaction structure of these classes. The Device Driver uses a client that is realized by an abstraction of the functional unit (an integral part of the MCU). A Peripheral Interface may directly implement one or multiple access policies. The following list shows some common access policies, but does not exclude others:

- Bit Access: Interface to set and clear (writing) or test (reading). Typically, a GPIO implements such pin-like behavior.
- Value Access: Interface to get (reading) a single value, like an ADC.
- Register Access: A register describes an addressable single value. Bus systems as I2C or SPI realize this policy that supports set and get.
- Character Access: Reading or writing an ordered sequence of unstructured characters (bytes). Serial interfaces as UART or a socket may implement such policy.

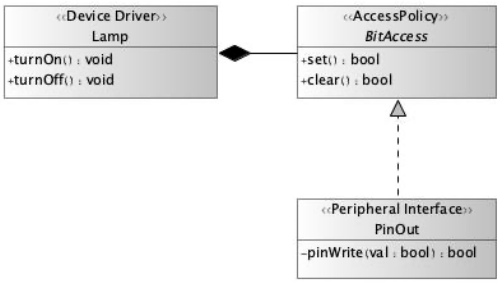

Figure 3. Lamp class requires bit access policy

Figure 3 shows the structure controlling a lamp that is externally connected via a GPIO (PinOut). The GPIO implements the Bit Access policy by using a pinWrite method. A more complex setup is the control of a BME280 sensor connected as an I2C device, shown in Figure 4. Since I2C is a bus allowing master-slave communication, we introduce an I2C Device Connector class as a link between the Device Driver and the Peripheral Interface.

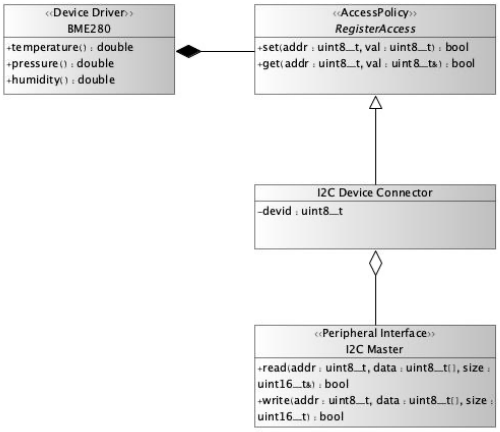

Figure 4. BME280 sensor connected via I2C

5.4. Consequences

Pros: By inverting the dependencies, the device drivers are independent of a concrete realization of the hardware and software platform. The portability, interchangeability as well as changeability, and thus

the reuse of higher software layers, is substantially improved. The use of the actual functionality of a device is detached from the concrete connection of the external devices and the configuration. Hereby, the approach fits into the use of tools for hardware configuration (e.g., provided by hardware vendors). The access policy limits the functional interface to a minimum.

Cons: The lowest hardware-related software layer must implement the access policy. Existing software libraries can apply the hardware adapter pattern [29] for this purpose. The effort is reduced since the access interface is kept to a minimum.

5.5. Implementation Strategies

In object-oriented programming, a policy can be realized by an interface class. In the C++ language, this is achieved by a class declaration with pure virtual methods. To avoid the disadvantages of this dynamic polymorphism, C++ offers alternatively the possibility of templates [30]. Our proposal is based on the standard C++20, where policies can be described generically using concepts as template parameters. Listing 1 shows the declaration for a pin-like access policy. A type T realizes the concept BitAccessable if it implements the methods set() and clear(). Both methods must return a boolean value.

```
/* --Pin-like Access. */
template<typename T>
concept BitAccessable = requires(T t) {
   { t.set()} -> std::same_as<bool>;
   { t.clear()} -> std::same_as<bool>;
};
```

Listing 1. Basic policy example

```
template<BitAccessable PIN>
class Lamp {
public:
  explicit Lamp(std::unique_ptr<PIN> p)
                      :_pin(std::move(p)) {}
  void turnOn() { _pin->set(); }
  void turnOff() { _pin->clear(); }
protected:

  /* --Pin to set/clear. */
  std::unique_ptr<PIN> _pin;
};
```

Listing 2. C++ class using a predefined policy

This concept describes write access to a pin, but can easily be adapted and extended to a read access with a method test(). A device class Lamp can then use this access strategy to turn a lamp on and off, as shown in Listing 2.

6. Energy-Aware Control Design

Compared to the current HAL implementations like the CMSIS [24], our approach is not limited to the MCU and directly connected interfaces. Furthermore, the existing energy-related functionalities are often unused and limited to on and off states as well as a not further defined low state for devices. Our approach is also vendor-independent and can be used for the complete system, including internal and external hardware devices, which can be proper classified by our approach.

This chapter sketches the concept of the vertical energy-aware control design. Section 6.1 gives a brief overview of the basic idea, while Section 6.2 presents the specified stereotypes for classes with the ability to control the power consumption of peripheral devices. Section 6.3 introduces our multilayered architecture and finally Section 6.4 presents design patterns for power consumption control.

6.1. Overview

Our aforementioned driver framework is extended to control the energy consumption of external devices according to the application requirements, while focusing on devices corresponding to the classifications 1 to 3 presented in Section 4. For this purpose, we first define a UML stereotype that outlines the basic properties of an energy-aware device.

In addition, we derive a three-layered architecture that provides functional access as well as energy consumption control. We integrate another two design patterns to coordinate the interaction of multiple devices, e.g., multiple sensors.

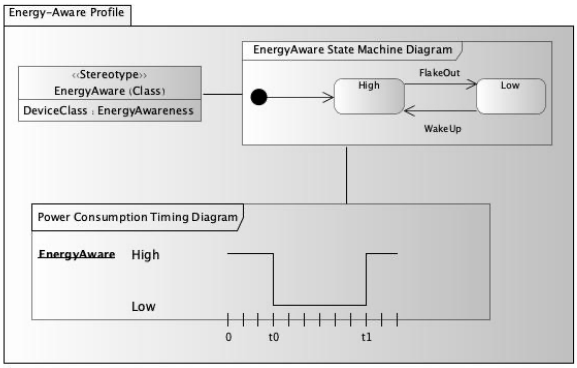

Figure 5. Power states defined by profile

6.2. Energy-Aware Stereotype

A class stereotyped as energy-aware has the ability to control the dynamic energy consumption of

a connected entity (see Figure 5). The energy consumption of a connected entity can be controlled by an algorithmic, software-based strategy (class 1 device) or by the exploitation of hardware properties (class 2 device). A combination of both methods is also possible (class 3 device).

The stereotype does not define an expected behavior but assumes that the energy-aware device has at least two different energy states (e.g., high and low). This simple state machine can be extended by adding sub-states to model complex energy-saving strategies. In any case, the specific behavior and energy consumption are defined by the device itself and not by the stereotype.

The change of an energy state is altered by two events, flake-out and wake-up. To allow different devices like sensors as well as actuators a dedicated behavior adapted to the current system state, events additionally transport a context describing the system state reflecting the constraint Con-2 (see Section 3).

6.3. Multilayered Architecture

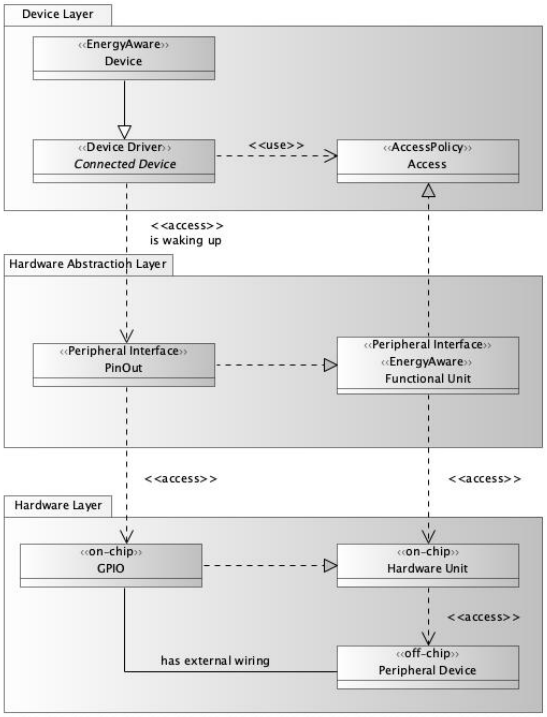

Figure 6. Architecture with dependency inversion

The presented device driver model is extended to include the stated energy-aware stereotype, leading to the layered architecture outlined in Figure 6. For better readability, the application layer is hidden. The lowest layer represents the physical hardware including the internal hardware units as well as the external peripheral devices (e.g., sensors), and consists of the following elements:

- Hardware Unit (on-chip): Integrated hardware components that implement specific device functionalities or communication protocols. The unit can be powered on and off (usually by clock gating). Examples of such units are UART and I2C.
- Peripheral Device (off-chip): A device that is physically connected to the MCU. The device may have internal power management functionalities.

The layer in the center of Figure 6 outlines the traditional HAL, which, however, has the inverted dependency on the device layer described at the beginning. The device driver layer implements functional access to the hardware and controls its energy consumption. The device layer consists of the following elements:

- Device: A concrete class, that represents a peripheral device. This class has to provide an interface to power management functionalities of the peripheral device and/or implements additional power management functions.
- Connected Device: Abstract class that is the foundation of concrete devices. A connected device requires a method to access the hardware and may have an additional pin-like interface to enable and disable the peripheral device.

6.4. Energy-Aware Control Pattern

Two design patterns are developed for controlling the power consumption of connected devices and are outlined below. These patterns have a different level of abstraction: the PowerLock design pattern is applied close to the hardware, while the PowerMonitor design pattern is located close to the application level.

The mechanism used by the proposed PowerLock design pattern controlling energy management is similar to the well-known procedures for mutual exclusion of a shared resource [31]. In analogy, this pattern is therefore referred to as PowerLock reflecting the constraint Con-4 (see Section 4). A PowerLock controls the energy state of a shared, energy-aware entity to ensure, that accessing the entity is possible from an energy perspective. The energy state is reduced when there is no access.

From an application point of view, the PowerMonitor design pattern [15] defines a sequential control path. During its execution, a set of energy-preserving entities is accessed. For this purpose, the corresponding energy states of the involved components are activated. From the energy point of view, access outside of the control path is not possible.

Multiple instances of a PowerMonitor can be nested to implement fine-granular control.

Figure 7 summarizes the main abstractions. The interface class PowerControlled is the basis for an entity that implements an energy-aware strategy. A realization of this class has to provide the power method. The two classes PowerLock and PowerMonitor control the access to a common resource in terms of energy consumption.

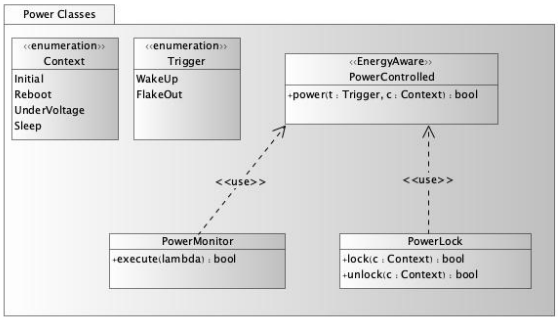

Figure 7. Energy consumption control interface

7. Evaluation

In the following, we describe the evaluation of the design patterns introduced above using a typical use case of an IoT device.

The proposed DIP variant is very much based on the use of concepts proposed in the C++20 standard. Although this standard and previous drafts have been available for some time, compilers do not yet sufficiently support all new functionalities. In particular, the migration of cross-compiler developments for embedded systems often shows delays of several years. We compare our implementation on two platforms: a typical desktop PC with C++20 support and a popular low-power MCU platform that is more widely used, especially in the IoT domain.

Besides evaluating the applicability of the presented method, it is also important to take the possibility and associated effort of integrating third-party source code into account. That includes the board support package of the platform and the drivers for external peripherals, both typically provided by hardware vendors.

Determining actual energy savings is very much dependent on the application and other factors such as the hardware platform used and the manufacturing quality. Through the measurement setup used, we have been able to reveal several energy bugs.

7.1. IoT-Use Case: Bee Colony Health Sensor

For the evaluation, we developed a sensor system to monitor the microclimate in western honeybee (apis mellifera) colonies. Beekeeping in magazines made of wood or polystyrene does not correspond to natural housing, which can have negative effects for bees. The humidity is actively influenced by the bees to ensure optimal environmental conditions for the colony and larvae [32].

The IoT system (pictured as an internal block diagram using the SysML notation in Figure 8), consists of two identical Bosch BME280 environmental sensors placed outside and inside the beehive, measuring temperature, air pressure, and humidity. The sensors are connected to the system via I2C and support different measurement modes. The RAK811 [33] is a communication module, which supports the Long Range Wide Area Network (LoRaWAN) specification [34] to transmit sensor data. The module is connected via UART and offers the possibility to switch to a sleep mode with reduced power consumption. In our use case, the system is defined as a LoRaWAN class A device, which is the most energy-efficient device class. To receive data, class A devices provide two download receive windows after each transmission phase. To indicate the system status to the beekeeper, an LED is connected via I2C. The current measurement is done with the Otii Arc device from Qoitech [35].

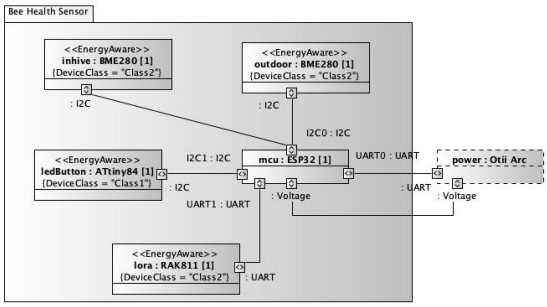

Figure 8. IoT application and measurement

To test the new possibilities in software design with C++20, the presented concept has been implemented initially on an x64-based PC using Microsoft Visual Studio 2019 (v16.9.4 with /std:c++latest), which supports the new syntax for concepts and additions to the C++ standard library. The hardware accesses are reduced to an emulation of the I2C bus. Further system dependencies are abstracted by using a software factory pattern. The required effort to migrate the application to the ESP32 platform [36] turned out to be negligible, although the cross compiler for the embedded platform (gcc v8.4.0 with –std=c++2a -fconcepts) does not implement the

current C++20 standard. The changes regarding the concepts could be limited to a few header files. As a result, the IoT application was quickly up and running, even though the energy savings were not immediately visible due to the energy bugs presented in Section 7.4.

7.2. Results

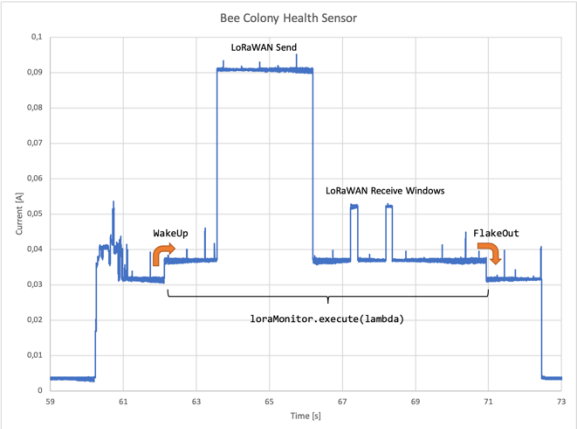

Figure 9. Current measurement of PowerMonitor

Figure 9 shows the power consumption while the system is in an active mode. The predominant part shows the sending of sensor data via LoRaWAN. Processing the transmit routine within the monitor automatically wakes up the RAK811 module and then resets it to sleep mode. It is only activated for the actually necessary time period.

```
/* --Sending sensor data... */
loraMonitor.execute([&](RAK811& lora) {
 bool ok=lora.send(_sendbuffer, size, 2);
 if (!ok)
  printf("Sending %i bytes failed",size);
 }
);
```

Listing 3. Lambda-expression enables LoRaWAN

Listing 3 shows the application-level view. The execute method of the PowerMonitor object is passed using a lambda expression. The underlying mechanisms are transparent to the software developer. The driver framework presented here in combination with energy-aware classes provides the link between hardware and application level and enables vertical control of energy consumption. A second step is to investigate the extent to which energy savings can be achieved applying the energy-aware framework.

The bee colony health sensor determines the climate data cyclically (75s period) and then switches to sleep mode.

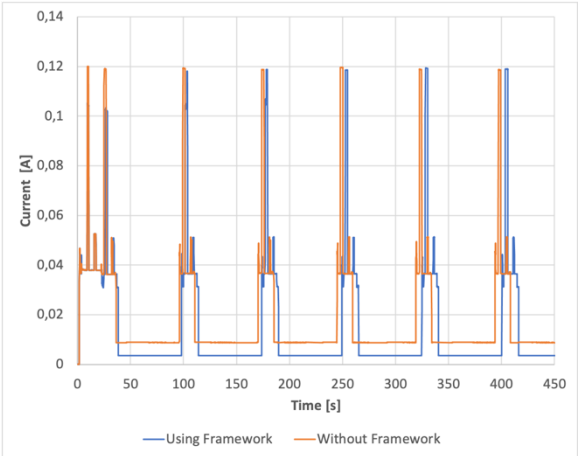

Figure 10. Comparison framework usage

Two measurement runs have been performed: with (i) and without (ii) use of the energy-aware framework. Figure 10 shows the total power consumption of the system (MCU including peripheral devices). The power consumption during sleep mode is significantly lower by using our framework. Furthermore, it takes a small amount of time to wake the peripherals up or put them into sleep mode. This extends the active phase of the IoT system and leads to a shift of the curve along the time axis.

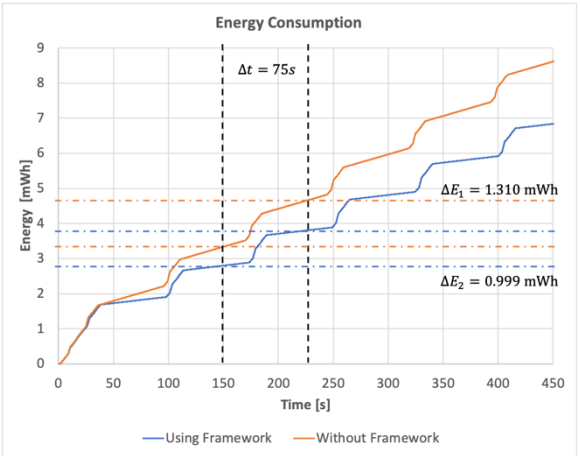

Figure 11. Energy savings

To determine the actual energy savings, a single cycle of active and sleep phases is examined. Figure 11 shows the accumulated energy consumption for the described use case. The energy savings $\Delta E_1 - \Delta E_2$ for a single cycle Δt using the explored framework is 0.311 mWh or approximately 23.7% of the total system consumption.

Compared to the environmental sensors, the energy consumption is significantly influenced by the LoRaWAN module in this example. The framework can always be applied successfully when individual

components are only used for a short time during a longer active phase of the system. This is the case with many IoT devices since the individual steps (sensor sampling, data processing, transmission) are completed sequentially.

7.3. Energy Bugs

In addition to the above results, there are other observations that can be classified as energy bugs. We list detected bugs here because they are independent of the developed software, occur for comparable environments, and can thus be helpful for developers.

First, the manufacturing quality of the hardware platform used has a significant influence on power consumption. Early prototype boards often use modular peripheral components (breakout boards), which may integrate additional LEDs to signal correct operations. The current consumption of a single LED (e.g., 1.6 mA) can distort or even mask the consumption of the actual peripheral component. The USB interface used for flashing can be a similar source of errors. Even when this interface is not used, a significant energy consumption can occur because of unoptimized hardware layouts.

A persistent error could be observed when the ESP32 system enters the deep sleep mode. Although the RAK811 was previously in sleep mode, the MCU forced the LoRa module to switch from sleep into an active mode. The detailed measurements performed revealed, the signal level on the UART interface was changed while the MCU was going into the deep sleep mode. The RAK811 module evaluates the changed level as a wake-up signal and thus increased the overall current consumption.

8. Summary and Outlook

Optimizations to increase the energy efficiency of an embedded system or IoT device often takes place at the hardware level. Software-hardware interactions are critical and it is often unclear how software affects the energy consumption of the system. Since software plays an important role when addressing energy efficiency, it needs careful consideration. In current HALs, an energy-aware peripheral control is only partially implemented and does not take specific properties of hardware components into account.

In this work, we proposed a new concept for an energy-aware peripheral control. Instead of defining abstractions for each hardware device, we used a more beneficial approach and proposed a classification containing four device classes with different requirements and properties according to energy management capabilities. For the energy-aware peripheral control, we presented a concept of a policy-based device pattern. The access policies described by the pattern define methods which are required by the hardware to be accessed by device drivers. By inverting the dependencies, device drivers are independent of a concrete realization of hardware interfaces. This improves the reusability and results in higher portability and interchangeability of the software. The proposed framework for energy-aware control design uses both, the classification and access policies to control hardware devices. All proposed concepts are aggregated into a three-layered software architecture, which provides functional access and energy consumption control coherently. We evaluated the software architecture in an IoT use case for hardware devices from different manufactures. The overall power consumption could be lowered without additional application code on higher levels.

Future work includes the extension of the energy-aware stereotypes for a more detailed description of energy-related properties of classes and the definition of additional access policies to be able to provide a more detailed access control. Furthermore, we want to evaluate the proposed approach in more complex use cases.

9. References

[1] B. H. Sudantha, E. J. Warusavitharana, G. R. Ratnayake, P. Mahanama, M. Cannata, and D. Strigaro, “Building an open-source environmental monitoring system - A review of state-of-the-art and directions for future research,” in *2018 3rd International Conference*, pp. 1–9.

[2] M. H. S. Nascimento, A. L. S. Moreira, M. A. O. Domingues, A. L. Castro, and D. L. Moura, “Implementing IoT for Development of a Low-Cost Environmental Monitoring System,” in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 122017, pp. 1360–1364.

[3] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, “Internet of Things for Smart Cities,” *IEEE Internet Things J.*, vol. 1, no. 1, pp. 22–32, 2014, doi: 10.1109/JIOT.2014.2306328.

[4] P. K. Illa and N. Padhi, “Practical Guide to Smart Factory Transition Using IoT, Big Data and Edge Analytics,” *IEEE Access*, vol. 6, pp. 55162–55170, 2018, doi: 10.1109/ACCESS.2018.2872799.

[5] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia, “An Overview of Internet of Things (IoT) and Data Analytics in Agriculture: Benefits and Challenges,” *IEEE Internet Things J.*, vol. 5, no. 5, pp. 3758–3773, 2018.

[6] L. Nobrega, A. Tavares, A. Cardoso, and P. Goncalves, “Animal monitoring based on IoT technologies,” in *2018 IoT Vertical and Topical*

Summit on Agriculture - Tuscany (IOT Tuscany), Tuscany, 52018, pp. 1–5.
[7] A. Gupta, T. Tsai, D. Rueb, M. Yamaji, and P. Middleton, *Forecast: Internet of Things: Endpoints and Associated Services, Worldwide, 2017.* [Online]. Available: https://www.gartner.com/en/documents/3840665/forecast-internet-of-things-endpoints-and-associated-ser (accessed: Sep. 29 2020).
[8] Cisco Systems, "Cisco Annual Internet Report (2018-2023)," Cisco C11-741490-01. Accessed: Sep. 29 2020.
[9] D. Rossi, I. Loi, A. Pullini, and L. Benini, "Ultra-Low-Power Digital Architectures for the Internet of Things," in *Enabling the Internet of Things: From Integrated Circuits to Integrated Systems*, M. Alioto, Ed., Cham: Springer International Publishing, 2017, pp. 69–93.
[10] G. Pinto, F. Castor, and Y. D. Liu, "Mining Questions about Software Energy Consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 22–31.
[11] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What Do Programmers Know about Software Energy Consumption?," *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016, doi: 10.1109/MS.2015.83.
[12] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "On testing embedded software," in *Advances in Computers*: Elsevier, 2016, pp. 121–153.
[13] K. Georgiou, S. Xavier-de-Souza, and K. Eder, "The IoT Energy Challenge: A Software Perspective," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 53–56, 2018.
[14] Object Management Group, *Unified Modeling Language, Version 2.5.1: OMG Document Number formal/17-12-05.* Accessed: Jun. 1 2021. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/
[15] M. Uelschen, M. Schaarschmidt, C. Fuhrmann, and C. Westerkamp, "PowerMonitor," in *Proceedings of the International Conference on Embedded Software Companion - EMSOFT '19*, New York, New York, 2019, pp. 1–2.
[16] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Investigating the effect of design patterns on energy consumption," *J Softw Evol Proc*, vol. 29, no. 2, e1851, 2017, doi: 10.1002/smr.1851.
[17] E. Gamma, J. Vlissides, R. Helm, and R. Johnson, *Design patterns: Elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.
[18] S. Maleki, C. Fu, A. Banotra, and Z. Zong, "Understanding the impact of object oriented programming and design patterns on energy efficiency," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, Orlando, FL, 2017, pp. 1–6.
[19] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides, "Energy Consumption Analysis Of Design Patterns," 2007, doi: 10.5281/zenodo.1057717.
[20] M. Buschhoff, R. Falkenberg, and O. Spinczyk, "Energy-aware device drivers for embedded operating systems," *SIGBED Rev.*, vol. 16, no. 3, pp. 8–13, 2019, doi: 10.1145/3373400.3373401.
[21] AUTOSAR, *Classic Platform.* [Online]. Available: https://www.autosar.org/standards/classic-platform/ (accessed: Jun. 1 2021).
[22] R. C. Martin and K. Henney, *Clean Architecture: A craftsman's guide to software structure and design*. Boston: Prentice Hall, 2018.
[23] IEEE and The Open Group, "The System Interfaces volume of POSIX.1-2017," Accessed: Jun. 1 2021. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition
[24] Arm Limited, *CMSIS-Driver: Peripheral Interface for Middleware and Application Code - Version 2.8.0.* [Online]. Available: https://arm-software.github.io/CMSIS_5/latest/Driver/html/index.html (accessed: Jun. 1 2021).
[25] Arduino, *Language Reference.* [Online]. Available: https://www.arduino.cc/reference/en/ (accessed: Jun. 1 2021).
[26] Arm Limited, *An introduction to Arm Mbed OS 6 - v6.11.* [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.11/introduction/index.html (accessed: Jun. 1 2021).
[27] Bosch Sensortec GmbH, "BME280 – Data sheet, Version 1.9: Document Number BST-BME280-DS001-18," BST-BME280-DS001-18. Accessed: Jun. 1 2021. [Online]. Available: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf
[28] A. Alexandrescu, *Modern C++ design: Generic programming and design patterns applied,* 2nd ed. Boston: Addison-Wesley, 2001.
[29] B. P. Douglass, *Design patterns for embedded systems in C,* 1st ed. Amsterdam: Newnes, 2011.
[30] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ templates: The complete guide*. Boston: Addison-Wesley, 2018.
[31] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Boston: Pearson Education, 2015.
[32] I. Eouzan *et al.,* "Hygroregulation, a key ability for eusocial insects: Native Western European honeybees as a case study," *PloS one*, vol. 14, no. 2, e0200048, 2019, doi: 10.1371/journal.pone.0200048.
[33] RAKwireless Technology Co., *RAK811-Module: Datasheet.* [Online]. Available: https://docs.rakwireless.com/Product-Categories/WisDuo/RAK811-Module/Datasheet/ (accessed: Jun. 7 2021).
[34] N. Sornin and A. Yegin, "LoRaWAN™ 1.1 Specification," 1.1. Accessed: Jun. 1 2021. [Online]. Available: https://lora-alliance.org/wp-content/uploads/2020/11/lorawantm_specification_-v1.1.pdf
[35] QOITECH, *Otii Arc.* [Online]. Available: view-source://https://www.qoitech.com/otii/ (accessed: Jul. 25 2021).
[36] Espressif Systems, *ESP32 Series: Datasheet*. V3.6. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (accessed: Jun. 1 2021).